UCRL-CONF-203489

LLNL's Parallel I/O Testing Tools and Techniques for ASC Parallel File Systems

W. E. Loewe, R. M. Hedges, T. T. McLarty, and C. J. Morrone

April 16, 2004

2004 IEEE Cluster Computing Conference, San Diego, CA, September 20-23, 2004

DISCLAIMER

LLNL'S Parallel I/O Testing Tools and Techniques for ASC Parallel File Systems

Bill Loewe
Richard Hedges
Tyce McLarty
Chris Morrone

Lawrence Livermore National Laboratory

April 16, 2004

ABSTRACT

Livermore Computing is an early and aggressive adopter of parallel file systems including, for example, GPFS from IBM and Lustre for our present Linux systems. As such, we have acquired more than our share of battle scars from encountering bugs in "bleeding edge" file systems that we have pressed into production to serve our customers' massive I/O requirements. A major role of the Scalable I/O Project is to detect errors before our end users do. In order to do this, we have developed highly parallel test codes to stress and probe potentially weak areas of file system behavior. This paper describes those test programs and how we make use of them.

Keywords: *MPI, File-I/O, Disk-I/O, Bandwidth, Testing, Performance.*

INTRODUCTION

The Department of Energy (DOE)'s Advanced Simulation & Computing (ASC) program has extreme high-performance computational demands that are ushering in increasingly bigger and faster computing systems for scientific research. To provide a means of transfer and storage for the correspondingly vast amounts of data generated as computational rates in excess of a petaflop are approached, high-performance parallel file systems are a critical system requirement.

The Scalable I/O Project (SIOP) at Lawrence Livermore National Laboratory (LLNL) supports ASC-scale applications generating many terabytes of data during an execution. We are engaged in testing the parallel file systems in all phases of their development in the hope that we will find problems before our users do and in working with application codes to ensure efficient I/O use of the parallel file system. An important element in the comprehensive support solution is to develop tools and techniques for testing the reliability and performance of the scalable file systems to meet the needs of the ASC program.

TESTING OBJECTIVES

The objective of our testing is to determine the stability, integrity, and performance of our parallel file systems. Our testing is used in the procurement phase of selecting a parallel file system to determine our future file system requirements. During the acceptance phase of bringing in a new parallel file system, we perform tests to determine whether a file system can pass its acceptance tests and fulfill its contact. We also investigate the behavior of the file system such that we may suggest improvements to our ASC partners developing the file system and consider approaches to the file systems' efficient use by our customers. And, we maintain regression testing of our parallel file systems to document performance and stability after system upgrades and changes as well as file system aging.

The level of stability is sought by putting a file system under load while scaling tasks accessing either shared or independent files. With shared files, all participating tasks have access to a single, common file, whereas with independent files, each task can only access its own unique file. In our experience, we have seen that though a parallel file system may handle low counts of multiple tasks for either of these approaches, significantly increasing the task count to production levels can uncover issues of stability and robustness.

The correctness and data integrity of a file system is absolutely expected by users. While a file system should be stable and rarely down or unavailable to users, the expectation that the data stored on it will not be corrupted is a fundamental requirement. This must be monitored closely, particularly in the development and stabilization phases of bringing in a new parallel file system. During these phases, the file system software is often temporarily broken in the process of forward progress.

Upon stabilizing a file system and finding it to be robust and correct, the inevitable concern for performance and tuning is addressed. We are interested in the performance of the file system operations involved with metadata activity and data movement. These performance rates can be calculated for a parallel file system using direct POSIX operations or through higher-level libraries such as MPI-IO, HDF5, and Parallel-NetCDF. The performance tests used for benchmarking are generally the most used of our testing tools.

With the goal of high-performance I/O for our users, further effort is then made in working with users' applications to ensure proper I/O models. For example, the higher-level libraries of MPI-IO, HDF5, or netCDF can improve the I/O approach for certain applications using large, noncontiguous datasets. A poor approach to I/O in an application can give even the best-performing file system substandard performance and become a bottleneck in the application's use. Unfortunately, I/O does not have high priority for code developers as they often feel the return on investment is marginal.

TESTING TOOLS

The SIOP uses several tools for its testing objectives. While such serial testing tools as *Bonnie++* [1], *dbench* [2], *fsx* [3], and *IOzone* [5] are useful, tools for parallel file systems need process synchronization for large task counts to ensure accurate testing. To aid us in testing parallel file systems, our project has developed the following parallel tools. *simul* [7] is used to test metadata integrity and stability by performing simultaneous metadata operations. *mdtest* [6] is used for testing and measuring metadata performance. *IOR* [4] is a workhorse that tests data performance rates for several access patterns and APIs.

simul

To test simultaneous file system operations from many nodes and processes, the application *simul* was developed. It is an MPI-coordinated test of parallel file system system calls and library functions that tests the correctness and coherence of a parallel file system. It accomplishes this by having one or more tasks perform standard POSIX operations to either shared or independent files.

In total, there are forty tests that *simul* performs to stress the simultaneous metadata operations. For both the shared and independent access modes, there are operations to files and directories. For files, the operations of open, close, file stat, lseek, read, write, unlink, rename, creat, truncate, symlink, readlink, link to one file, and link to file-per-process are performed. For directories, the chdir, directory stat, readdir, mkdir, and rmdir operations are tested. *simul* provides a high instantaneous load in metadata operations, providing a rigorous test of a parallel file system.

mdtest

mdtest allows multiple MPI tasks in parallel to create, stat, and remove shared or independent files. The rate of these operations is reported by *mdtest* and can be used to determine if file system performance problems are related to metadata performance.

For example, *mdtest* was used on a large parallel file system at LLNL to test whether the observed sluggishness of the file systems was due to aging as it filled near capacity. It appeared that as the parallel file system grew to over 90% full, the time to it took to allocate inodes for creating multiple files seemed to slow down to the point where it was a tenth of what it was previously capable. *mdtest* allowed the diagnosis and understanding of the file system slow down, helping us to determine the issues with the metadata performance.

mdtest has several parameters that may be set in testing. It has the option to perform its tests with tasks working in the same or independent directories. Also, it allows the number of file or directories that each task accesses to be set which can simulate the behavior of various user application codes. Further, it has an option to prevent reading locally-cached file or directory information before performing a stat on a newly-created file by using a 'read-your-neighbor' node approach. In addition, *mdtest* has other settings that may be used to test metadata performance behavior.

IOR

IOR is a file system bandwidth testing code developed and used at LLNL. Employing MPI for process synchronization, *IOR* can be used for testing bandwidth performance of parallel file systems using various interfaces and access patterns. These interfaces and patterns try to simulate the file system used byASC applications. In our arsenal of testing tools, *IOR* is the most heavily used for a high, sustained load on a parallel file system.

Its interfaces include the POSIX, MPI-IO, HDF5, and Parallel-NetCDF APIs. Though many ASC applications use the POSIX API for their I/O, some codes are beginning to employ the capabilities of the high-level libraries. *IOR* attempts to follow the native access patterns inherent in each of these interfaces.

IOR has the capability of either shared or independent file access. As the task count for parallel applications increases, using a file-per-process approach becomes untenable due to the large number of files. Attempting to list a directory containing thousands of files becomes a challenge. To alleviate this overhead of large numbers of files, users' applications are beginning to follow a single, shared file paradigm. *IOR* offers this access to test the performance of a shared file access.

IOR also checks for data integrity and reports incidences and locations of data corruption. Upon writing or reading, a file check may be performed on the file to determine any corruption during the write or read phase of file access. In order that the performance results are not skewed, the write and read data checks are not performed during the data transfer measurements. Further, to assure that stored rather than cached data is checked, there is an option to have a different process perform the check than had performed the data transfer. These checks have been useful in finding many problems in the file system.

Named for the acronym from 'interleaved or random', *IOR* allows for different access patterns to a shared file. The primary distinction between the two major shared-file patterns is whether each task's data is contiguous or noncontiguous. For the segmented pattern, each task (*A* and *B*, e.g.) stores its blocks of data in a contiguous region in the file. This *aaabbb* pattern is generally less challenging to a file system as larger data buffers can be transferred and there are fewer requests/revocations of locks for byte-ranges in the file. With the strided access pattern, each task's data blocks are spread out through a file and are noncontiguous. While this *ababab* pattern may be more natural for storing a multi-dimensional mesh, it can be inefficient from the viewpoint of the file system due to additionallocking and smaller data transfers.

For the MPI-IO, HDF5, and Parallel-NetCDF interfaces, the access may be either collective or independent. With collective operations, multiple tasks participate in a file operation gathering I/O requests into a single request. With independent operations, each task’s I/O request is independent of the others’ requests. *IOR* allows both the collective and independent operations for the I/O libraries to be exercised, helping to determine which model may be more effective for a particular case. Generally, the collective model offers the greatest benefit for the strided access pattern wherein data from multiple tasks may be aggregated into a larger access. For the segmented pattern in large files, the collective access may serve only to add communication overhead, and the independent I/O model can be more efficient.

Among the other features of the testing code is the ability to pass hints to the MPI-IO library and make ioctl() calls as necessary to specific file systems. Also, *IOR* may be executed by command line for ease in scripting or with a GUI for interactivity. One of its most important features is its ability to be modified quickly for an immediate, one-off test. The code has become a powerful “Swiss Army knife” with more features available than mentioned here.

In addition to testing the file system behavior and simulating the applications’ I/O access patterns, *IOR* has even been used to find bugs in I/O library implementations. For example, in an implementation of the MPI I/O library, *IOR* helped to diagnose a problem in accessing MPI elemental datatypes (etypes). The MPI standard declared that units for counting file offsets should be in etypes, but the implementation was using only the byte unit for calculating all offsets as done in POSIX. *IOR* noticed that the finished file size was inaccurate for the test and was able to help track down the cause.

TESTING TECHNIQUES

As the goal of the parallel file system is to keep I/O from being in the way of the applications, approaches to understanding the interaction of the file system and the user code have been considered. With the understanding gained, the SIOP can suggest improvements to the file system design as application needs are clarified and also uncover inefficient I/O models in the user code that are inherently difficult for a file system.

A simple technique to monitor all I/O activity on an I/O node of a parallel file system is with UNIX’s *iostat*. It can be used to determine the load on an I/O device to find any bottlenecks and diagnose file system behavior. For example, were the I/O device showing a high block write or read rate, it would be reasonable to assume that the file system was working near capacity and only slowed by the physical hardware. But, were this rate lower than expected, the bottleneck in performance would likely be elsewhere, either in the file system design or the application I/O usage.

For a detailed look into an application’s I/O calls, Linux’s *strace* or AIX’s *truss* can be used. These tools provide a trace of all system calls and their parameters. While the amount of trace data can be quite large, the I/O behavior of an application can be learned by data mining the output. As an example of *strace*'s use by our project, an application we looked into with a code developer turned up thousands of 32-byte write calls. It had not been clear that the application was making so many and such small transfers. This type of inefficient I/O behavior can reduce file system performance significantly.

In another situation where a particular user's code was triggering file system errors, we used *strace* to estimate input parameters for *IOR* to model that code's I/O behavior. We knew that the total file size was ~20 GB, that 49 processes were active in the I/O, and that the transfers were interleaved from the processes. The tracing showed that most of the data was transferred in ~1.8 MB transfers. That information was sufficient to specify a set of input parameters to the *IOR* program in order to model the user's I/O. When we were able to reproduce the file system error with the *IOR* test code, we had considerably simplified subsequent testing of that problem in the following ways: 1) The bug could now be demonstrated with a test program which was well know to the test and development teams; 2) The problems in dealing with any sensitivities of the original program or resultant data were eliminated.

ISSUES

CACHE

One of the major issues affecting our strategy in testing a file system is how to account for caching. In particular, as the Linux OS is aggressive in its use of cache, obtaining accurate measurements for write/read performance or reliable results for a data check can be challenging. For example, data that is expected to have been written to storage media is often cached locally or on the device's cache. Actually syncing all the data to storage can be difficult.

We have used different approaches in our attempts to defeat cache. The first approach works for local cache. Quite simply, it is to reduce the amount of memory the operating system can access. By setting `mem=512M`, for example, much of the unused memory that the cache can use is inaccessible. A drawback, however, is that this approach requires rebooting the nodes with the lower setting for available memory. What is gained is to allow the write performance to be closer to the bandwidth to storage than it is to the memory bandwidth. Since the cache is small, testing the read performance after the write performance is more accurate, too.

Another approach is to have a neighbor node read back the data such that the data written is only on the original node's cache, not on the neighbor node. This "read your neighbor" approach works well for reads, but it does little for the cached data on writes.

Since we usually cannot reset the available memory and reboot a node on a production system, and we are generally cannot access and reduce the cache on the device (be they I/O nodes, object storage targets, etc.), the most common approach to reducing the effect of caching is to increase the test file size to the point where the data cannot be cached both locally and remotely. While this works, it has the disadvantage of increasing the test time by moving more data. Since testing time is always limited, our testing becomes less comprehensive and more likely to miss file system problems. Therefore, handling caching incorrectly can have an impact on file system performance testing.

Of course the value of these approaches to considering the issue of caching is debatable as well. One line of thought is that if the testing code is designed to simulate the application's I/O behavior and caching improves the performance, then the testing code using caching is capturing the true performance as used by the application. While this has merit, the testing code and the application are likely to use significantly different amounts of memory, thereby having different remaining cache sizes available. To disable caching to the extent that it is possible levels the playing field.

POSIX vs. MPI IO

The data stream model of a file used by the POSIX standard can deliver very high performance if the access pattern used by the application looks like a stream of data. However many applications, especially parallel ones, access data in smaller chunks in a strided pattern through the file. I/O performance with this type of access pattern is usually much slower than the file system capability unless some means is used to coordinate the data flow to or from the file system. System caching can reduce the strided effect in some cases, but if each or most I/O operations cause a disk seek, caching is of little help.

MPI-IO provides a way to do the data coordination for a parallel application in a standard, portable way. When MPI datatypes and collective operations are used, a much greater fraction of the file system capability is delivered by aggregating smaller chunks of data using the high speed communication structure and doing larger, well-aligned I/O operations. The *IOR* program includes an MPI-IO option so that the data coordination capabilities of a particular MPI-IO implementation can be tested systematically.

FUTURE WORK

In general, our I/O work is driven by the problems and issues found as they present themselves. This can range from procurement to regression testing and from file system debugging to user application issues. Despite having a tendency to be interrupt-driven, however, our project also makes sure to preemptively prepare ourselves for coming issues as we foresee them. This can include impending file system design changes and improving our knowledge of how applications perform I/O.

CONCLUSION

The parallel testing tools developed in the SIOP offer a significant improvement in scaling testing of ASC parallel file systems over the existing serial tests. As we are on the forefront of parallel file system testing, our understanding of the issues of these file systems for large clusters is still expanding and new problems themselves are surfacing. The tools and techniques we have thus far developed are an excellent start to solving some of the problems in parallel file system testing, but only a beginning.

The testing tools *IOR*, *mdtest*, and *simul* have been released under the GNU General Public License and can be found at:

http://www.llnl.gov/icc/lc/siop/downloads/download.html

Our project website is located at:

http://www.llnl.gov/icc/lc/siop/

ACKNOWLEDGMENTS

We thank Brian Behlendorf, Ryan Braby, Matt Davis, Dave Fox, Terry Heidelberg, Linda Stanberry, and Andrew Uselton for their design considerations, contributions, usage, bug fixes, feedback, and sage advice with these testing tools and strategies.

AUSPICES STATEMENT

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

REFERENCES

[1] Bonnie++ [2001]. http://www.coker.com.au/bonnie++/.

[2] dbench [2001]. http://freshmeat.net/projects/dbench.

[3] fsx [1991]. http://www.codemonkey.org.uk/cruft/fsx-linux.c.

[4] IOR [2003]. # UCRL-CODE-2003-016.
http://www.llnl.gov/icc/lc/siop/downloads/download.html.

[5] IOzone [2003]. http://www.gnu.org/directory/sysadmin/Monitor/Iozone.html.

[6] mdtest [2003]. # UCRL-CODE-155800.
http://www.llnl.gov/icc/lc/siop/downloads/download.html.

[7] simul [2003]. # UCRL-CODE-2003-019.
http://www.llnl.gov/icc/lc/siop/downloads/download.html.